LIBXSMM
LIBRARY TARGETING INTEL ARCHITECTURE (X86)
FOR SMALL, DENSE OR SPARSE MATRIX MULTIPLICATIONS, AND SMALL CONVOLUTIONS.

Hans Pabst
Intel High Performance and Throughput Computing
Switzerland

Alexander Heinecke
Parallel Computing Lab
Intel Labs
USA

Greg Henry
SSG Pathfinding
USA

March 9th 2017

https://github.com/hfp/libxsmm
LIBXSMM, for small, dense or sparse matrix multiplications, and small convolutions on Intel Architecture

https://github/hfp/libxsmm, Hans Pabst, Alex Heinecke, and Greg Henry, Intel

General-purpose code cannot be optimal:

- If all cases are supported, the library is too large, too much branching, etc.
- Lack of specialization hurts when matrices are 1-10 SIMD units on a side.

Runtime specialization captures best of both worlds:

- “Perfect” code for only the cases needed; unused code is never generated.
- Just-in-Time (JiT) compilation for general code is hard.
- Specific domain (SMM, DNN, etc.) allows for JiT code generation without a compiler.
LIBXSMM Function Domains

Main function domains in LIBXSMM

- **SMM**: Small Matrix Multiplication Kernels (original library)
- **DNN**: Deep Neural Network Kernels for CNNs (v1.5)
- **SPMDM**: Sparse Matrix Dense Matrix Multiplication for CNNs (v1.6)
- **AUX**: Mem. alloc., synchronization, debugging, profiling

There is more functionality…

- Tiled GEMM routines based on SMM kernels (also parallelized)
- Stand-alone out-of-place matrix transpose routines (non-JIT, soon JIT)
- Matrix-copy kernels (JIT)
- Other “sparse routines”
LIBXSMM: Overview

Highly efficient Frontend
- BLAS compatible (DGEMM, SGEMM) including LD_PRELOAD
- Support for F77, C89/C99, F2003, C++
- Zero-overhead calls into assembly
- Two-level code cache

Code Generator
- Supports all Intel Architectures since 2005, focus on AVX-512
- Prefetching across small GEMMs
- Can generate assembly (*.s), inline assembly (*.h/*.c), and in-memory code

Just-In-Time (JIT) Encoder
- Encodes instructions based on basic blocks
- Very fast code generation (no compilation)
Primer about GEMM…

**GEneral Matrix Matrix** routines for single, and double-precision

Original call/arguments:  
\[
\text{DGEMM('N', 'N', M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)} \rightarrow \text{Fixed/bound at: Static compilation-time}
\]

JIT: only a subset of Alpha/Beta values is supported (can be exploited for optimization)

JIT-GEMM descriptor: \(M, N, K, LDA, LDB, LDC, \text{“Flags”}, \text{and “Prefetch”}

**Flags**: TransA & TransB, **Prefetch**: strategy (implies arity of JIT-code)
LIBXSMM: GEMM Descriptor

Two descriptor sizes are available: Optimized and Default/BIG

- Implies whether only a subset of problems is supported or not
- Optimized descriptor size (16 Byte): SSE-enabled dispatch
- Default/BIG descriptor: scalar dispatch-code might be faster

Dispatch flow: descriptor → CRC32 hash/index → [cache] → code registry

- Registry “hit” requires at least one comparison (two descriptors)
- Due to the hash, collisions must handled as well
- Note: [cache] is skipped here for clarity
```c
#include <libxsmm.h>

int main()
{
    const double alpha = 1.0, beta = 1.0;
    const int m = 23, n = 23, k = 23;  /* some problem size */
    double a[m*k], b[k*n], c[m*n];    /* init. not shown! */
    libxsmm_dmmfunction xmm = NULL;   /* function pointer */

    libxsmm_gemm(NULL, NULL, &m, &n, &k,
                  &alpha, a, NULL, b, NULL,
                  &beta, c, NULL);

    /* like function interface for low-level JIT’ted kernel */
    libxsmm_dmm_23_23_23(a, b, c);    /* specialized */

    xmm = libxsmm_dmmdispatch(23, 23, 23, NULL, NULL, NULL,
                                &alpha, &beta, NULL, NULL);
    if (xmm) {
        /* specialized */
        for (int i = 0; i < some; ++i) {
            xmm(a, b, c);               /* amortized */
        }
    }
}
```

LIBXSMM (C API): Example
Application

GEMM

Frontend
User API for C/C++ and Fortran, call interception (static linkage, and LD_PRELOAD), and code dispatch

Call
(80 ≤ $\sqrt[3]{MNK}$)

Check Difference

$3\sqrt{MK} \leq 128$

Binary Blob (Hash / Diff.) consists of:
TRANSA, TRANSB, M, N, K, LDA, LDB, LDC, ALPHA, BETA

Call

Check

Threshold

Receive / Call

Codeversion

Thread-local Code Cache

Check

Hash / Difference

Generate / Store

Code Registry

Backend for JIT code (via API) or statically generated code (driver program, which prints C code with inline assembly)

Fallback (BLAS)
LIBXSM: Code Registry

Custom data structure and algorithm for fast code retrieval

- Arbitrary capacity (size could be dynamic, but is fixed at compile-time)
- Capacity exploits Prime number (Mersenne) properties to simplify bitops

Custom thread-safety

- Atomic reads are used, and locks are only used to protect updates/writes
- Multiple locks (POT number to simplify bitops) used to protect writes
  Multiple updates are allowed for different locations
- Code duplication is possible due to readers not participating in locks
  Mainly happens because of hash key collisions and multiple writer-locks
  Only happens when code generation request is contended
LIBXSMM: Code Cache

Properties

- Plays well with optimized descriptor size (multiple of SIMD width, etc.)
- Small capacity (default: 4 entries)
- Fully associative cache
- LRU-style eviction

Purpose

- Accelerate calls via GEMM interface (a.k.a. auto-dispatched)

Cache Primer

- **Direct Mapped Cache**: simplest form of cache, check for a hit without search (only one possible place which may hold an address). Drawback: same cache location shared for many addresses (depends on cache / memory size ratio).
- **Fully Associative Cache**: best hit ratio since any cache location can hold any address. Drawback: expensive search needed.
- **N-Way Set Associative Cache**: addresses fall into bins of fixed capacity; search needed within hit-bin. Drawback: compromise.
Evaluation of suitable JIT code generators

- Numerous projects evaluated: jitasm, libgccjit, etc.
- Selection/rejection criterions
  - Support for recent Intel Architectures,
  - Active development
- Interesting candidates (with comments)
  - LLVM Full-blown (with IR, phases, etc.), “slow” JIT, complex
  - Xbyak compiler and JIT-assembler, incomplete AVX-512 (2015!)
  - XED Closed source (2016: https://github.com/intelxed/xed)

Final decision in 2015: own development needed “JIT Assembler”

- Only “a few” instructions needed for a certain domain (still true)
- No legacy support needed (AVX/2 and beyond is fine)
LIBXSMN Backend: Runtime Code Generation (Very High Level Idea)

Idea: leveraged GNU Compiler extension “Computed GOTO”

```
LABEL1:
c = a + b;
LABEL2:

memcpy(code, &LABEL1, &LABEL2 - &LABEL1);
```

Reality: LIBXSMN manually encodes all instructions needed

- Basic form is encoded with placeholder(s) for varying parts (immediates)
- Emitting an instruction: call a function (arguments may cover instruction variants and/or immediates), to write a whole kernel is like using a DSL (“assembly programming domain”)
Quick facts about in-memory JIT code generation (JIT assembler)

- No intermediate representation
- No automatic register allocation
- No (compiler-)optimizations

What is the advantage of JIT code?

- It is able to leverage instruction variants/immediates to hardcode runtime knowledge (hard to statically compile equivalent code!)
  
  Example: hard-coded stride for load instruction address (broadcast ld.)

- Why is there a particular focus on AVX-512? There is a lot of potential in the instruction set e.g., EVEX may also encode certain values into instruction
LIBXSMM AVX512 code for N=9

- Column-major storage; working on all 9 columns and 8 rows simultaneously
- Loads to A (vmovapd) are spaced out to cover L1$ misses; K-loop is fully unrolled
- B-elements are broadcasted within the FMA instruction to save execution slots (SIB)
- SIB addressing mode to keep instruction size <= 8 byte for 2 decodes per cycle (16 byte I-fetch per cycle)
- Multiple accumulators (zmm31-xmm23 and zmm22-zmm14) for hiding FMA latencies

→ Max. theoretical efficiency: 90%! 
LIBXSMM Backend: JIT Overhead (incl. OS calls)

- Xeon E5-2697v4 - JIT compile time in microseconds
- Xeon E5-2697v4 - JIT compile time in MKL calls

JIT compile time in microseconds

JIT compile time in MKL calls

M=N=K
LIBXSMM Performance on 1c Xeon E5-2697v4 (BDX)

- LIBXSMM, static
- Intel MKL 11.3.2, direct-call
- Eigen-3.3-beta1, ICC 16.0.2, dynamic
- Eigen-3.3-beta1, GCC 4.9.2, dynamic
- BLAZE 2.6, ICC 16.0.2, dynamic
- BLAZE 2.6, GCC 4.9.2, dynamic
- LIBXSMM, JIT
- Eigen-3.3-beta1, ICC 16.0.2, static
- Eigen-3.3-beta1, GCC 4.9.2, static
- BLAZE 2.6, ICC 16.0.2, static
- BLAZE 2.6, GCC 4.9.2, static
- PEAK

GFLOPS (DP) vs. M=N=K

2 4 6 8 10 12 14 16 18 20

0 4 8 12 16 20 24 28 32
LIBXSMM Performance on 1c Xeon Phi 7250 (KNL)

- LIBXSMM, static
- Intel MKL 11.3.2, direct-call
- Eigen-3.3-beta1, ICC 16.0.2, dynamic
- Eigen-3.3-beta1, GCC 4.9.2, dynamic
- BLAZE 2.6, ICC 16.0.2, dynamic
- BLAZE 2.6, GCC 4.9.2, dynamic
- LIBXSMM, JIT
- Eigen-3.3-beta1, ICC 16.0.2, static
- Eigen-3.3-beta1, GCC 4.9.2, static
- BLAZE 2.6, ICC 16.0.2, static
- BLAZE 2.6, GCC 4.9.2, static
- PEAK (38.4 Gflops for 1 core @1.2 GHz)
LIBXSMM vs. MKL DGEMM_BATCH
2x Xeon E5-2697v4 (BDX)

- BDX - LIBXSMM
- BDX - MKL 11.3.2 (BATCHED)
- BDX - LIBXSMM bandwidth
LIBXSMM: Developments

Support for “medium-sized” and “big” matrix multiplication

• Tiled matrix multiplication routines to go beyond SMM

• OpenMP for multicore support
  • Thread-based multicore with internal parallel region
  • Task based when called from parallel region

• Planned: support for Eigen-library and non-OpenMP MT

Initial support for stand-alone matrix transposes

• Tiled transpose optionally with task-based OpenMP
LIBXSMM: Other Features

CPUID-dispatched (critical) code paths

Makes LIBXSMM suitable for Linux distributions where the code path (target system) is unpredictable (1 package)

Link-time and Runtime Wrapper

Intercepts existing xGEMM calls at runtime (LD_PRELOAD) or at link-time (LD’s --wrap)

JIT Profiling (Intel VTune Amplifier)

https://github.com/hfp/libxsmm/#profiling

The "function name" is supplied by LIBXSMM using VTune’s JIT Profiling API

libxsmm_hsw_dnn_23x23x23_23_23_23_a1_b1_p0::jit

- Encodes an Intel AVX-512 ("knl") double-precision kernel ("d") which is multiplying matrices without transposing them ("nn"),
- Rest of the name encodes M=N=K=LDA=LDB=LDC=23, Alpha=Beta=1.0 (all similar to GEMM),
- No prefetch strategy ("p0").
LIBXSMM DNN

* https://github.com/hfp/libxsmm/#interface-for-convolutions
DNN API for Convolutional Neural Networks (CNNs)

- Introduced in LIBXSMM 1.5, refined in v1.6 and v1.7
- Features:
  - Fallback code, JIT-code: AVX2, and AVX-512 (Common, Core, KNM)
  - Forward convolution, backward convolution, and weight transformation
  - Data formats: NHWC, RSCK, and custom format
  - Compute types: f32, and i16 (+ i8 as in-/output)
- Major additions in v1.8: logical padding, Winograd, KNM

Quick summary
- Handle-based API to generate/perform the requested transformation
LIBXSMM DNN API

- Sample code (samples/dnn) to also act as benchmark for convolutions

- Results
  

TensorFlow


- Initial integration only in master revision of TensorFlow

- Scheduled for TF 1.1
LIBXSMM: Applications

[1] http://cp2k.org/: Open Source Molecular Dynamics with its DBCSR component processing batches of small matrix multiplications ("matrix stacks") out of a problem-specific distributed block-sparse matrix. Starting with CP2K 3.0, LIBXSMM can be used to substitute CP2K's 'libsmm' library. Prior to CP2K 3.0, only the Intel-branch of CP2K was integrating LIBXSMM (see https://github.com/hfp/libxsmm/raw/master/documentation/cp2k.pdf).

[2] https://github.com/SeisSol/SeisSol/: SeisSol is one of the leading codes for earthquake scenarios, in particular for simulating dynamic rupture processes. LIBXSMM provides highly optimized assembly kernels which form the computational back-bone of SeisSol (see https://github.com/TUM-I5/seissol_kernels/).

[3] https://github.com/Nek5000/NekBox: NekBox is a version of the highly scalable and portable spectral element Nek5000 code which is specialized for box geometries, and intended for prototyping new methods as well as leveraging FORTRAN beyond the FORTRAN 77 standard. LIBXSMM provides optimized kernels aiming to conveniently substitute the MXM_STD code.


[5] https://tensorflow.org/: TensorFlow™ is an open source software library for numerical computation using data flow graphs. TensorFlow was originally developed by researchers and engineers working on the Google Brain Team for the purposes of conducting machine learning and deep neural networks research. LIBXSMM can be used to increase the performance of TensorFlow on Intel hardware.
LIBXSMM Performance (CP2K)

* https://github.com/hfp/libxsmm/tree/results#libxsmm-results
Molecular World View and Computational Approach

A metaphorical depiction of how to improve upon the treatment of electron correlation by ascending from the Hartree world to the “heaven of chemical accuracy.”

CP2K implements most if not all of the methods!
CP2K and DBCSR (Small Matrix Multiplications)

CP2K implements for instance* the Density Functional Theory (DFT) among other methods.

- DFT can be seen as a general Eigenvalue problem, which is solved using the Self-Consistent Field (SCF) iterative method.
- Sparsity can be exploited, and ends up with small dense blocks of natural structure (atoms).
- Recent CPUs (FMA) are doing very well.

Distributed Blocked Compressed Sparse Row
Distributed Blocked Cannon Sparse Recursive

- DBCSR* library is ubiquitously used by almost all algorithms in CP2K (not only for DFT).
- DBCSR generates matrix batches (“stacks”) of ~1000 small matrices: C += A * B (accumulation).

* Pictures adapted from Speedup 2012 (Joost VandeVondele)

* CP2K’s sparse matrix library: https://dbcsr.cp2k.org/
CP2K and DBCSR (Small Matrix Multiplications)

- DFT can be seen as general Eigenvalue problem, which is solved using the Self-Consistent Field (SCF) iterative method
- Sparsity can be exploited, and ends up with small dense blocks of natural structure (atoms)
- Recent CPUs (FMA) are doing very well

Most if not all methods in CP2K rely on the SpBLAS-like DBCSR component, and produce batches of small matrix multiplications.

(Nobel Prize 1998: Walter Kohn and John A. Pople)

- DBCSR* library is ubiquitously used by almost all algorithms in CP2K (not only for DFT)
- DBCSR generates matrix batches (“stacks”) of ~1000 small matrices: \( C += A \times B \) (accumulation)

* Pictures adapted from Speedup 2012 (Joost VandeVondele)

* CP2K’s sparse matrix library: https://dbcsr.cp2k.org/
What’s the Problem with SMMs?

- 1162 “relevant” DP kernels are shown in the right picture
- All relevant kernels show some low (DP-)arithmetic intensity
- Exploiting (MC)DRAM bandwidth is key for performance
- A reproducer needs to actually stream the A and B matrices!
CP2K Kernel Performance for DBCSR Library

Dual Socket Xeon E5-2697v4

Single Socket Xeon Phi 7250
LIBXSMM Performance Results – SeisSol

Intel Xeon E5-2699v3 (“HSW”)
Intel Xeon E5-2670 (“SNB”)
Intel Xeon X5690 (“WSM”)
Intel Xeon Phi 5110P (“KNC”)
Seismic Wave Propagation & Dynamic Rupture Simulation

Initial Fault Stresses

Geological Structure
(Fault Geometry & Material Properties)

SeisSol

Ground Shaking
(Seismograms), Fault Split, etc.

- Full elastic wave equations in 3D and complex heterogeneous media
- Dynamic Rupture without artificial oscillations
- High order: ADER(time)-DG(space)
- Unstructured tetrahedral meshes
- Highly optimized kernels, massively parallel (multi PFLOPS, GB14 finalist)
- Local Time Stepping

(Brietzke et al. (2009))
SeisSol’s Compute Kernels

Small Matrix-Matrix multiplications, (convergence order 6): 9x9, 56x9, 56x35

A priori known sparsity patterns
SeisSol LOH.1 Benchmark Performance

Dual Socket Xeon E5-2697v4

- LIBXSMM, nzGFLOPS
- LIBXSMM, pGFLOPS
- Intel MKL 11.3.2, nzGFLOPS
- Intel MKL 11.3.2, pGFLOPS
- LIBXSMM, bandwidth
- Intel MKL 11.3.2, bandwidth

Single Socket Xeon Phi 7250

- LIBXSMM, nzGFLOPS
- LIBXSMM, pGFLOPS
- Intel MKL 11.3.2, nzGFLOPS
- Intel MKL 11.3.2, pGFLOPS
- LIBXSMM, bandwidth
- Intel MKL 11.3.2, bandwidth
NekBox

High Order CFD Simulations

Joint Work with Maxwell Hutchinson (U Chicago), David Keyes (KAUST)

ISC’16 paper
NekBox

NekBox is a stripped-down of Nek5000 for box-shaped domains

NekBox solves the incompressible Navier-Stokes equations:

\[
\frac{\partial u}{\partial t} + u \cdot \nabla u = -\frac{1}{\rho} \nabla p + \nu \nabla^2 u + f \quad \nabla \cdot u = 0
\]

Incl. advection-diffusion equations for scalar variables such as temperature or mass fractions.

Nek uses the spectral element method (SEM):

- A tensor product of Gauss-Lobatto-Legendre (GLL) quadrature points within each element -> \(N^3\) DOFs per element -> small GEMM
- Continuity across elements -> forming a mesh (using direct stiffness summation)
- NekBox comes also with its own mxmin_std library (FORTRAN)

Operators are written as element local operators:

\[
A = (A_x \times I_y \times I_z) + (I_x \times A_y \times I_z) + (I_x \times I_y \times A_z)
\]

which reduces the complexity from \(O(N^6)\) to \(O(N^4)\).
NekBox’s main compute routines

A typical NekBox run spends <1% in sparse computations & communications, ~40% in vector-vector or matrix-vector operations, ~60% matrix-matrix operations.

Helmholtz operator:

```plaintext
Hu(:, :, :) = gx(:, :, :) * matmul(Kx(:, :,)), reshape(u, (/N, N*N/)))
do i = 1, n
   Hu(:, :, i) += gy(:, :, i) * matmul(u(:, :, i), KyT(:, :))
endo
Hu(:, :, :) += gz(:, :, :) * matmul(reshape(u, (/N*N, N/)), KzT(:, :))
Hu(:, :, :) = h1 * Hu(:, :, :) + h2 * M(:, :, :) * u(:, :, :)
```

Basis transformation:

```plaintext
tmp_x = matmul(Ax, u)
do i = 1, n
   tmp_y(:, :, i) = matmul(tmp_x(:, :, i), AyT)
endo
v = matmul(tmp_y, AzT)
```

Gradient calculation:

```plaintext
dudx = matmul(Dx, u)
do i = 1, n
   dudy(:, :, i) = matmul(u(:, :, i), DyT)
endo
dudz = matmul(u, DzT)
```

→ Batched GEMM is not beneficial as it would mean losing locality
NekBox Small GEMM operations

- **GigaFLOPS (DP)**
- **GB/s**

Various libraries and hardware configurations are compared:
- **KNL - LIBXSMM**
- **KNL - Intel MKL 11.3.2**
- **KNL - mxm_std**
- **BDX - LIBXSMM**
- **BDX - Intel MKL 11.3.2**
- **BDX - mxm_std**

The chart illustrates performance metrics across different operators and configurations.
NekBox Performance on a Xeon Phi XC40

- NekBox; Nek5000-derivate for boxed domains
- Simulation of secondary flows in square ducts
- Important application areas are turbomachinery and heat exchangers, where rectangular ducts and diffusers are ubiquitous
- More than 1 PFLOPS sustained end-to-end performance on Cray XC40 (9216 nodes with dual socket Intel® Xeon® E5-2698v3)
Summary
Multi-Node Performance (SeisSol, NekBox, CP2K)

- SeisSol - LIBXSMM
- SeisSol - Intel MKL 11.3.2
- NekBox - LIBXSMM
- NekBox - mxm_std
- CP2K - LIBXSMM
- CP2K - LIBSMM
- CP2K - MKL 2017u1

Perf. relative to smallest node count, and relative to LIBXSMM

Number of Nodes of Xeon E5v4

Performance graph showing the relative performance of SeisSol, NekBox, and CP2K across different node counts and software versions.
Conclusion

• Small GEMMs are an important kernel in real science applications
• They could be batched, but for best performance leveraging CPU features such as fast caches is essential
• Using LIBXSMM, we showed several factors in performance for small GEMMs compared to vendor BLAS or compile-time optimized libraries at kernel level
• At full application level and at scale up an to 50% performance increase is possible, this is even much higher in case of Xeon Phi where optimal code is highly important due to the architecture
• Current research is applying the proposed technology to direct convolutions for deep learning (11 input parameters instead of 6), early implementation is already available on GitHub for AVX2 and AVX512.

https://github.com/hfp/libxsmm
LIBXSMM: References


LEGAL DISCLAIMER & OPTIMIZATION NOTICE

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804